

// Computer Program Listing Appendix Under 37 CFR 1.52(e)

// Attribute.java

// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.

```
package com.sybase.djc;
import com.sybase.djc.compiler.*;
import com.sybase.djc.util.*;
import java.lang.reflect.*;
import java.util.*;
public abstract class Attribute
{
    private String _fieldName;
    private String _methodName;
    private ParameterList _parameters;
    public AttributeCompiler getCompiler()
    {
        return null;
    }
    public boolean matches(Field field)
    {
        return _fieldName != null
            && field.getName().equals(_fieldName);
    }
    public boolean matches(Method method)
    {
        return _methodName != null
            && method.getName().equals(_methodName)
            && matchParameterTypes(method);
    }
    public boolean overrides(Attribute that)
    {
        if (this.getClass() != that.getClass())
        {
            return false;
        }
        else if (this._methodName != null)
        {
            if (that._methodName != null)
            {
                if (this._methodName.equals(that._methodName))
                {
                    if (this._parameters == null)
                    {
                        return that._parameters == null;
                    }
                    else
                    {
                        return this._parameters.hasSameTypes(that._parameters);
                    }
                }
            }
            else
            {
                return false;
            }
        }
        else if (that._methodName != null)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
}
```

```

    {
        return false;
    }
}
else
{
    return true;
}
}
else if (that._methodName != null)
    {
        return false;
    }
    else if (this._fieldName != null)
    {
        if (that._fieldName != null)
        {
            return this._fieldName.equals(that._fieldName);
        }
        else
        {
            return true;
        }
    }
else if (that._fieldName != null)
{
    return false;
}
    else
    {
        return true;
    }
}
public boolean okIfMethodNotFound()
{
return false;
}
public int getSortOrder()
{
    return 0;
}
public void setFieldName(String fieldName)
{
    _fieldName = fieldName;
}
public String getFieldName()
{
    return _fieldName;
}
public void setMethodSignature(String methodSignature)

```

```

{
    int paren = methodSignature.indexOf("(");
    if (paren != -1)
    {
        String params = methodSignature.substring(paren + 1).trim();
        _methodName = methodSignature.substring(0, paren).trim();
        _parameters = getParameterList(params);
    }
    else
    {
        // Any parameters will match.
        _methodName = methodSignature;
        _parameters = null;
    }
}

public String getMethodName()
{
    return _methodName;
}

public String getMethodSignature()
{
    if (_methodName == null)
    {
        return "";
    }
    String sig = _methodName;
    if (_parameters != null)
    {
        sig += _parameters.toStringWithTypes();
    }
    return sig;
}

public ParameterList getParameterList()
{
    return _parameters;
}

public String toString()
{
    String a = StringUtil.removeSuffix(JavaClass.getNameSuffix(getClass().getName()), "Attribute");
    String s;
    if (_methodName != null)
    {
        s = a + "(" + getMethodSignature() + ")";
    }
    else if (_fieldName != null)
    {
        s = a + "(" + getFieldName() + ")";
    }
    else
    {

```

```

        s = a + "()";
    }
    Method[] methods = getClass().getMethods();
    int n = methods.length;
    for (int i = 0; i < n; i++)
    {
        Method method = methods[i];
        if (method.getReturnType() == getClass())
        {
            String getMethodName = "get" + StringUtil.getUpperFirst(method.getName());
            try
            {
                Method getMethod = getClass().getMethod(getMethodName, new Class[0]);
                Object value = getMethod.invoke(this, new Object[0]);
                if (value != null && value instanceof String)
                {
                    value = "" + value.toString() + "";
                }
                if (value != null)
                {
                    s += "." + method.getName() + "(" + value + ")";
                }
            }
            catch (Exception ignore)
            {
            }
        }
    }
    return s;
}

private ParameterList getParameterList(String params)
{
    ParameterList list = new ParameterList();
    if (params.endsWith("("))
    {
        params = params.substring(0, params.length() - 1).trim();
    }
    StringTokenizer st = new StringTokenizer(params, ",");
    for (int p = 0; st.hasMoreTokens(); p++)
    {
        String token = st.nextToken().trim(), type, name;
        int space = token.indexOf(' ');
        if (space == -1)
        {
            type = token;
            name = "p" + (p + 1);
        }
        else
        {
            type = token.substring(0, space).trim();

```

```

        name = token.substring(space + 1).trim();
    }
    if (type.length() == 0 || name.length() == 0)
    {
        throw new SystemException("Bad Method Parameter List: " + _methodName + "(" + params + ")");
    }
    list.add(type, name);
}
return list;
}
private boolean matchParameterTypes(Method method)
{
    if (_parameters == null)
    {
        return true;
    }
    Class[] types = method.getParameterTypes();
    if (types.length != _parameters.size())
    {
        return false;
    }
    int p = 0;
    for (Iterator i = _parameters.iterator(); i.hasNext(); p++)
    {
        MethodParameter mp = (MethodParameter)i.next();
        String type = types[p].getName();
        if (! type.equals(mp.type))
        {
            if (! type.endsWith(".") + mp.type))
            {
                return false;
            }
        }
    }
    return true;
}
}
// AttributeCompiler.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
package com.sybase.djc.compiler;
import java.lang.reflect.*;
import com.sybase.djc.*;
import com.sybase.djc.util.*;
public abstract class AttributeCompiler
{
    private static final String COMPONENT_STATIC_FIELD = "$djc_component";
    public java.util.List addInterfaces()
    {
        return null;
    }
}

```

```

public void compile(Component component, ClassWriter cw)
{
}
public void compile(Component component, Field field, ClassWriter cw)
{
}
public void abstractBody(Component component, Method method, MethodWriter mw)
{
}
public void beforeSuperCall(Component component, Method method, MethodWriter mw)
{
}
public void afterSuperCall(Component component, Method method, MethodWriter mw)
{
}
public String getComponent(Component component)
{
    return "com.sybase.djc.Component.forClass("
        + component.getBaseClass().getName() + ".class)";
}
public String getComponentField(Component component, CodeWriter cw)
{
    if (!cw.hasStaticField(COMPONENT_STATIC_FIELD))
    {
        cw.newStaticField(Component.class, COMPONENT_STATIC_FIELD, getComponent(component))
            .setPrivate();
    }
    return COMPONENT_STATIC_FIELD;
}
public boolean ignoreMethod(Component component, Method method)
{
    if (method.getName().startsWith("$"))
    {
        return true;
    }
    for (Class bc = component.getBaseClass(); bc != null; bc = bc.getSuperclass())
    {
        if (bc.getName().equals("javax.servlet.http.HttpServlet"))
        {
            String sig = JavaMethod.getShortSignature(method);
            if (!sig.equals("service(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServletResponse)")
                && !sig.equals("init(javax.servlet.ServletConfig)")
                && !sig.equals("destroy()"))
            {
                return true;
            }
        }
    }
    return false;
}

```

```

    public void setJavaCompiler(JavaCompiler jc)
    {
    }
}
// Component.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
package com.sybase.djc;
import com.sybase.djc.attributes.*;
import com.sybase.djc.compiler.*;
import com.sybase.djc.properties.*;
import com.sybase.djc.repository.*;
import com.sybase.djc.util.*;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
public class Component
{
    // inner classes
    private static class PackageInfoKey
    {
        private ClassLoader _classLoader;
        private String _infoClass;
        private PackageInfoKey(ClassLoader classLoader, String infoClass)
        {
            _classLoader = classLoader;
            _infoClass = infoClass;
        }
        public int hashCode()
        {
            int h = _classLoader == null ? 0 : _classLoader.hashCode();
            return h ^ _infoClass.hashCode();
        }
        public boolean equals(Object thatObject)
        {
            PackageInfoKey that = (PackageInfoKey)thatObject;
            return this._classLoader == that._classLoader
                && this._infoClass.equals(that._infoClass);
        }
    }
    // public data
    public static final int AllMethods = 1;
    public static final int ObjectMethods = 2;
    public static final int ProtectedMethods = 3;
    public static final int PublicMethods = 4;
    // private data
    private static final String PACKAGE_INFO_SUFFIX = ".PackageInfo";
    private static final Class[] EMPTY_CLASS_ARRAY = {};
    private static final Object[] EMPTY_OBJECT_ARRAY = {};
    private static boolean _bootstrap = SystemProperties.bootstrap();
    private static boolean _recompile = SystemProperties.recompile();

```

```

private static boolean _debug = SystemProperties.debug();
private static boolean _quiet = SystemProperties.quiet();
private static boolean _verbose = SystemProperties.verbose();
private static boolean _batchStaticAttributesGeneration = SystemProperties.batchStaticAttributesGeneration();
private static boolean _batchDynamicAttributesGeneration =
SystemProperties.batchDynamicAttributesGeneration();
private static HashMap _components = new HashMap();
private static HashMap _packages = new HashMap();
private static HashMap _mockObjectMap = new HashMap();
private static boolean _checkMockObjects = false;
private static boolean _resetAllAttributes = false;
private Class _baseClass;
private String _componentClassName;
private FutureObject _componentClass = new FutureObject()
{
public Object evaluate()
{
return evaluateComponentClass();
}
};
private boolean _sharedObject;
private FutureObject _singleton = new FutureObject()
{
public Object evaluate()
{
return evaluateSingleton();
}
};
private Component _packageInfo;
private File _propertiesFile;
private PropertyMap _properties;
private AttributeList _attributes;
private HashMap _compilerMap;
private boolean _hasAttributesField;
private boolean _hasComponentField;
private boolean _initAgainWasCalled;
private SortOrderAttribute _sortOrder;
private List _methods;
private HashMap _methodIdMap;
private boolean _compiled;
private boolean _loaded;
// public methods
public Component(Class baseClass)
{
synchronized (_components)
{
_components.put(baseClass, this);
}
init(baseClass);
}

```



```

    public static Component forClass(Class baseClass)
    {
    if (baseClass == null)
    {
        throw new IllegalArgumentException("baseClass is null");
    }
    Component component;
    Field componentField;
        try
        {
            try
            {
                componentField = baseClass.getField("$component");
            }
            catch (Exception ignore)
            {
                componentField = baseClass.getField("component");
            }
            component = (Component)componentField.get(null);
        }
        catch (NoSuchFieldException ex)
        {
    if (_debug)
    {
        System.out.println("DEBUG: new Component: " + baseClass.getName());
    }
    synchronized (_components)
    {
        component = (Component)_components.get(baseClass);
        if (component == null)
        {
            component = new Component(baseClass);
            // constructor adds to _components.
        }
    }
    return component;
        }
        catch (Exception ex)
        {
            throw new SystemException("component " + baseClass.getName(), ex);
        }
    if (component == null)
    {
        throw new SystemException(componentField.getName()
            + " field is null in class " + baseClass.getName());
    }
    return component;
    }
    public static Component forName(String className)
    {

```

```

        return forClass(ThreadContext.loadClass(className));
    }
    public Object getInstance()
    {
        if (_checkMockObjects)
        {
            Object mockObject = _mockObjectMap.get(_baseClass);
            if (mockObject != null)
            {
                return mockObject;
            }
        }
        if (_sharedObject)
        {
            return getSingleton();
        }
        else
        {
            return newInstance();
        }
    }
    public Object newInstance()
    {
        Class cc = getComponentClass();
        try
        {
            return cc.newInstance();
        }
        catch (RuntimeException ex)
        {
            throw (RuntimeException)ex;
        }
        catch (Exception ex)
        {
            throw new SystemException(ex);
        }
    }
    public Object getNamedInstance(String instanceName)
    {
        try
        {
            Method getInstance = _baseClass.getMethod("getInstance", new Class[] { String.class });
            Object instance = getInstance.invoke(instanceName, new Object[] { instanceName });
            return instance;
        }
        catch (SystemException ex)
        {
            throw ex;
        }
        catch (Exception ex)

```

```

{
    throw new SystemException(ex);
}
}
public boolean hasAttributesField()
{
    return _hasAttributesField;
}
public String toString()
{
    return "component " + getName();
}
// private methods
private void init(Class baseClass)
{
    _baseClass = baseClass;
    if (_componentClassName == null)
    {
        _componentClassName = getName() + "_DJC";
    }
    _attributes = new AttributeList();
    loadProperties();
    initPackageInfo();
    if (_packageInfo != null)
    {
        add(_attributes, _packageInfo.getAttributes());
    }
    initAttributes(baseClass);
    if (!_hasAttributesField)
    {
        add(_attributes, getStaticAttributes());
    }
    add(_attributes, getDynamicAttributes());
    if (_hasComponentField && (baseClass.getModifiers() & Modifier.ABSTRACT) == 0)
    {
        if (!_bootstrap && !BootstrapObject.class.isAssignableFrom(baseClass))
        {
            ComponentLog.getInstance(_baseClass).warnClassIsNotAbstract();
        }
    }
    _sharedObject = _attributes.findAttribute(SharedObjectAttribute.class) != null;
}
private void initAttributes(Class c)
{
    Class sc = c.getSuperclass();
    if (sc != null && sc != Object.class)
    {
        Component scc = Component.forClass(sc);
        scc.initAgain();
        _attributes.addAll(scc.getAttributes());
    }
}

```

```

    }
    Class[] interfaces = c.getInterfaces();
    for (int i = 0; i < interfaces.length; i++)
    {
        sc = interfaces[i];
        Component scc = Component.forClass(sc);
        scc.initAgain();
        _attributes.addAll(scc.getAttributes());
    }
    Field[] fields = c.getDeclaredFields();
    int n = fields.length;
    for (int i = 0; i < n; i++)
    {
        Field field = fields[i];
        Class fieldType = field.getType();
        String fieldName = field.getName();
        if (fieldType == Component.class)
        {
            if (!fieldName.equals("$component")
                && !fieldName.equals("component"))
            {
                continue;
            }
            if ((field.getModifiers() & Modifier.FINAL) == 0)
            {
                if (!_bootstrap)
                {
                    ComponentLog.getInstance(_baseClass).warnFieldIsNotFinal(field.getName());
                }
                continue;
            }
            if ((field.getModifiers() & Modifier.PUBLIC) == 0)
            {
                if (!_bootstrap)
                {
                    ComponentLog.getInstance(_baseClass).warnFieldIsNotPublic(field.getName());
                }
                continue;
            }
            if ((field.getModifiers() & Modifier.STATIC) == 0)
            {
                if (!_bootstrap)
                {
                    ComponentLog.getInstance(_baseClass).warnFieldIsNotStatic(field.getName());
                }
                continue;
            }
            _hasComponentField = true;
        }
        else if (fieldType == Attribute[].class)

```

```

{
    if (! fieldName.equals("$attributes")
        && ! fieldName.equals("attributes"))
    {
        continue;
    }
    if ((field.getModifiers() & Modifier.FINAL) == 0)
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldIsNotFinal(field.getName());
        }
        continue;
    }
    if ((field.getModifiers() & Modifier.PUBLIC) == 0)
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldIsNotPublic(field.getName());
        }
        continue;
    }
    if ((field.getModifiers() & Modifier.STATIC) == 0)
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldIsNotStatic(field.getName());
        }
        continue;
    }
    try
    {
        Attribute[] attributes = (Attribute[])field.get(null);
        _hasAttributesField = true;
        if (attributes != null)
        {
            for (int j = 0; j < attributes.length; j++)
            {
                Attribute a = attributes[j];
                add(_attributes, a);
            }
        }
        catch (Exception ex)
        {
            ex.printStackTrace();
        }
    }
}
_sortOrder = SortOrderAttribute.getDefault();

```

```

    }
    private void initPackageInfo()
    {
        if (getName().endsWith(PACKAGE_INFO_SUFFIX))
        {
            _packageInfo = null;
            return;
        }

        String javaPackage = JavaClass.getPackagePrefix(getName());
        if (javaPackage.length() == 0)
        {
            _packageInfo = null;
            return;
        }

        String nameSuffix = JavaClass.getNameSuffix(getName());
        String infoClass = javaPackage + PACKAGE_INFO_SUFFIX;
        PackageInfoKey piKey = new PackageInfoKey(_baseClass.getClassLoader(), infoClass);
        synchronized (_packages)
        {
            Object pi = _packages.get(piKey);
            if (pi == Boolean.FALSE)
            {
                _packageInfo = null;
                return;
            }
        }
        _packageInfo = (Component)pi;
        if (_packageInfo == null)
        {
            try
            {
                Class piClass = loadClass(infoClass);
                _packageInfo = Component.forClass(piClass);
                _packages.put(piKey, _packageInfo);
            }
            catch (Exception ex)
            {
                boolean ignore = false;
                if (ex instanceof SystemException)
                {
                    Throwable cause = ((SystemException)ex).getCause();
                    if (cause != null && cause instanceof ClassNotFoundException)
                    {
                        ex = (Exception)cause;
                    }
                }
                if (ex instanceof ClassNotFoundException)
                {
                    ClassNotFoundException cnfe = (ClassNotFoundException)ex;
                    String message = cnfe.getMessage();
                    if (message != null && message.equals(infoClass))

```

```

    {
        ignore = true;
    }
}
if (! ignore && ! _bootstrap)
{
    ComponentLog.getInstance(_baseClass).warnPackageInfoNotFound(ex);
}
_packageInfo = null;
_packages.put(piKey, Boolean.FALSE);
}
}
}
private boolean isPackageInfo()
{
return getName().endsWith(PACKAGE_INFO_SUFFIX);
}
private AttributeList getStaticAttributes()
{
if (! _hasComponentField)
{
return null;
}
if (SystemProperties.batchStaticAttributesGeneration())
{
if (! _resetAllAttributes)
{
return null;
}
}
}
String acName = getName() + "$SA";
boolean generate = _recompile && ! _batchStaticAttributesGeneration;
if (BootstrapObject.class.isAssignableFrom(_baseClass))
{
// Only generate static attributes for bootstrap components
// in bootstrap mode (and not for bootstrap child processes).
if (! _bootstrap || SystemProperties.bootstrapChild())
{
generate = false;
}
}
if (generate)
{
// Generate static attributes class from source code.
File jsFile = JavaClass.getSourceFile(_baseClass);
if (jsFile != null)
{
File acFile = JavaClass.getClassFile(acName);
File bcFile = JavaClass.getClassFile(_baseClass);

```

```

        if (acFile == null
            || (bcFile != null && bcFile.lastModified() >= acFile.lastModified()))
        {
            StaticAttributes.generate(jsFile.getPath());
        }
    }
}
Class ac = null;
try
{
    ac = loadClass(acName);
}
catch (Exception notFound)
{
    if (_debug)
    {
        notFound.printStackTrace();
    }
    if (ac != null)
    {
        return getAttributes(ac);
    }
else
{
    return null;
}
}
private AttributeList getDynamicAttributes()
{
    if (! _hasComponentField && ! isPackageInfo())
    {
        return null;
    }
    if (SystemProperties.batchDynamicAttributesGeneration())
    {
        if (! _resetAllAttributes)
        {
            return null;
        }
    }
    String acName = getName() + "$DA";
    boolean generate = _recompile && ! _batchDynamicAttributesGeneration;
    if (BootstrapObject.class.isAssignableFrom(_baseClass))
    {
        // Only generate dynamic attributes for bootstrap components
        // in bootstrap mode (and not for bootstrap child processes).
        if (! _bootstrap || SystemProperties.bootstrapChild())
        {
            generate = false;

```



```

    }
}
if (_bootstrap || _baseClass == Repository.class)
{
    generate = false;
}
if (generate)
{
    // Generate dynamic attributes class from properties.
    DynamicAttributes.generate(this);
}
Class ac = null;
try
{
    ac = loadClass(acName);
}
catch (Exception notFound)
{
    if (_debug)
    {
        notFound.printStackTrace();
    }
    if (ac != null)
    {
        return getAttributes(ac);
    }
else
{
    return null;
}
}
private AttributeList getAttributes(Class c)
{
    AttributeList list = new AttributeList();
    Field[] fields = c.getDeclaredFields();
    int n = fields.length;
    for (int i = 0; i < n; i++)
    {
        Field field = fields[i];
        Class fieldType = field.getType();
        String fieldName = field.getName();
        if (fieldType != Attribute[].class)
        {
            continue;
        }
        if (! fieldName.equals("attributes"))
        {
            continue;
        }
    }
}

```

```

    if ((field.getModifiers() & Modifier.FINAL) == 0)
    {
        if (!_bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldIsNotFinal(field.getName());
        }
        continue;
    }
    if ((field.getModifiers() & Modifier.PUBLIC) == 0)
    {
        if (!_bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldIsNotPublic(field.getName());
        }
        continue;
    }
    if ((field.getModifiers() & Modifier.STATIC) == 0)
    {
        if (!_bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldIsNotStatic(field.getName());
        }
        continue;
    }
    try
    {
        Attribute[] attributes = (Attribute[])field.get(null);
        for (int j = 0; j < attributes.length; j++)
        {
            Attribute a = attributes[j];
            add(list, a);
        }
    }
    catch (Exception ex)
    {
        throw new SystemException(ex);
    }
}
return list;
}
/**
 ** Add attribute to list. If the attribute has the same class as an
 ** existing attribute, remove the existing attribute (the new one
 ** overrides the existing one, e.g. method-level overrides class-level).
 **/
private void add(AttributeList list, Attribute toAdd)
{
    if (toAdd == null)
    {
        return;
    }

```

```

    }
    if (toAdd instanceof SortOrderAttribute)
    {
        _sortOrder = (SortOrderAttribute)toAdd;
        return;
    }
    if (toAdd instanceof AttributeList)
    {
        // Flatten attribute hierarchy
        for (Iterator i = ((AttributeList)toAdd).iterator(); i.hasNext();)
        {
            Attribute nested = (Attribute)i.next();
            add(list, nested);
        }
        return;
    }
    for (Iterator i = list.iterator(); i.hasNext();)
    {
        Attribute existing = (Attribute)i.next();
        if (toAdd.overrides(existing))
        {
            list.remove(existing);
            break;
        }
    }
    list.add(toAdd);
}

public String getName()
{
    return _baseClass.getName();
}

public Class getBaseClass()
{
    return _baseClass;
}

public Component getPackageInfo()
{
    return _packageInfo;
}

public AttributeList getAttributes()
{
    return _attributes;
}

public static Class[] getComponents(Class packageInfo)
{
    try
    {
        Field componentsField = packageInfo.getField("components");
        int modifiers = componentsField.getModifiers();
        if ((modifiers & Modifier.PUBLIC) == 0)
    }

```

```

    {
        throw new SystemException("in class " + packageInfo.getName() + ", field 'components' is not public");
    }
    if ((modifiers & Modifier.STATIC) == 0)
    {
        throw new SystemException("in class " + packageInfo.getName() + ", field 'components' is not static");
    }
    Class type = componentsField.getType();
    if (type != Class[].class)
    {
        throw new SystemException("in class " + packageInfo.getName() + ", field 'components' has wrong type");
    }
    Class[] components = (Class[])componentsField.get(null);
    if (components.length == 0)
    {
        if (!_bootstrap)
        {
            ComponentLog.getInstance(packageInfo).warnPackageHasNoComponents();
        }
    }
    return components;
}
catch (SystemException ex)
{
    throw ex;
}
catch (Exception ex)
{
    if (_debug)
    {
        ex.printStackTrace();
    }
    if (!_bootstrap)
    {
        ComponentLog.getInstance(packageInfo).warnPackageHasNoComponentsField();
    }
    return new Class[0];
}
}

public static void resetAllAttributes()
{
    _resetAllAttributes = true;
}

public void initAgain()
{
    if (_resetAllAttributes && !_initAgainWasCalled)
    {
        init(_baseClass);
        _initAgainWasCalled = true;
    }
}

```

```

}
public AttributeList getAttributes(Method method)
{
    return getAttributes(method, null);
}
public PropertyMap getProperties()
{
    return _properties;
}
public boolean quiet()
{
    return getProperties().getProperty("quiet", "false").equals("true");
}
public boolean verbose()
{
    return getProperties().getProperty("verbose", "false").equals("true");
}
public synchronized HashMap getCompilerMap()
{
    if (_compilerMap == null)
    {
        _compilerMap = new HashMap();
    }
    return _compilerMap;
}
private void loadProperties()
{
    if (_bootstrap
        || _baseClass == com.sybase.djc.log.LogManager.class
        || _baseClass == com.sybase.djc.log.Logger.class
        || _baseClass == Repository.class
        || _baseClass == RepositoryLog.class)
    {
        _properties = new PropertyMap();
        return;
    }
    if (getName().endsWith(PACKAGE_INFO_SUFFIX))
    {
        _properties = Repository.getInstance().getPackageProperties(JavaClass.getPackagePrefix(getName()));
    }
    else
    {
        _propertiesFile = Repository.getInstance().getComponentPropertiesFile(_baseClass);
        if (!_propertiesFile.exists())
        {
            Repository.getInstance().setComponentProperties(_baseClass, new PropertyMap());
        }
        _properties = Repository.getInstance().getComponentProperties(_baseClass);
    }
}
}

```

```

/**
 ** Get all the attributes which are (potentially) applicable to a
 ** specified method, ordered by the current sort order.
 **/
private AttributeList getAttributes(Method method, MethodWriter mw)
{
    AttributeList list = new AttributeList();
    for (Iterator i = _attributes.iterator(); i.hasNext();)
    {
        Attribute a = (Attribute)i.next();
        if (a.getMethodName() == null)
        {
            boolean okToAdd = true;
            for (Iterator j = list.iterator(); j.hasNext();)
            {
                Attribute b = (Attribute)j.next();
                if (b.overrides(a))
                {
                    okToAdd = false;
                    break;
                }
            }
            if (okToAdd)
            {
                add(list, a);
            }
            else if (a.matches(method))
            {
                add(list, a);
                if (a.getParameterList() != null && mw != null)
                {
                    // Method writer must adopt parameter names from attribute.
                    for (Iterator j = mw.getParameterList().iterator(), k = a.getParameterList().iterator(); j.hasNext();)
                    {
                        MethodParameter mp = (MethodParameter)j.next();
                        MethodParameter ap = (MethodParameter)k.next();
                        mp.name = ap.name;
                    }
                }
            }
        }
    }
    list = list.sort(_sortOrder);
    return list;
}

public static void setMockObject(Class baseClass, Object mockObject)
{
    _mockObjectMap.put(baseClass, mockObject);
    _checkMockObjects = true;
}

```

```

public Field getField(String fieldName)
{
    return getField(fieldName, _baseClass);
}
private Field getField(String fieldName, Class bc)
{
    if (bc == null)
    {
        return null;
    }
    try
    {
        return bc.getDeclaredField(fieldName);
    }
    catch (NoSuchFieldException notFound)
    {
        return getField(fieldName, bc.getSuperclass());
    }
}
public List getMethods()
{
    if (_methods == null)
    {
        _methods = getMethods(_baseClass);
    }
    return _methods;
}
public Method getMethod(String methodName, Class[] parameterTypes)
{
    ParameterList p1 = new ParameterList(parameterTypes);
    for (Iterator i = getMethods().iterator(); i.hasNext();)
    {
        Method method = (Method)i.next();
        if (method.getName().equals(methodName))
        {
            ParameterList p2 = new ParameterList(method);
            if (p1.hasSameTypes(p2))
            {
                return method;
            }
        }
    }
    return null;
}
public int getMethodID(Method method)
{
    return getMethodID(method.getName(), method.getParameterTypes());
}
public synchronized int getMethodID(String methodName, Class[] parameters)
{

```

```

    if (_methodIdMap == null)
    {
        _methodIdMap = new HashMap();
    }
    String sig = methodName + new ParameterList(parameters).toStringWithTypesOnly();
    Integer id = (Integer)_methodIdMap.get(sig);
    if (id != null)
    {
        return id.intValue();
    }
    id = new Integer(_methodIdMap.size() + 1);
    _methodIdMap.put(sig, id);
    return id.intValue();
}

public boolean match(Method method, int methods)
{
    switch (methods)
    {
        case AllMethods:
            return true;
        case ObjectMethods:
            return isObjectMethod(method);
        case ProtectedMethods:
            if ((method.getModifiers() & Modifier.PROTECTED) != 0
                && ! isObjectMethod(method))
            {
                return true;
            }
            break;
        case PublicMethods:
            if ((method.getModifiers() & Modifier.PUBLIC) != 0
                && ! isObjectMethod(method))
            {
                return true;
            }
            break;
    }
    return false;
}

public static String showMethods(int methods)
{
    switch (methods)
    {
        case Component.AllMethods:
            return "AllMethods";
        case Component.ObjectMethods:
            return "ObjectMethods";
        case Component.ProtectedMethods:
            return "ProtectedMethods";
        case Component.PublicMethods:

```



```

        return "PublicMethods";
    default:
        throw new IllegalArgumentException("methods = " + methods);
    }
}

public static boolean isObjectMethod(Method method)
{
    try
    {
        Object.class.getDeclaredMethod(method.getName(), method.getParameterTypes());
        return true;
    }
    catch (NoSuchMethodException notFound)
    {
        return false;
    }
}

public ClassWriter getClassWriter()
{
    PropertyMap props = null;
    Component packageInfo = null;
    TreeMap addInterfaces = new TreeMap();
    for (Iterator i = _attributes.iterator(); i.hasNext(); )
    {
        Attribute a = (Attribute)i.next();
        AttributeCompiler ac = a.getCompiler();
        if (ac != null)
        {
            List addList = ac.addInterfaces();
            if (addList != null)
            {
                for (Iterator j = addList.iterator(); j.hasNext(); )
                {
                    Object c = j.next();
                    if (c instanceof Class)
                    {
                        Class ci = (Class)c;
                        String cs = ci.getName();
                        addInterfaces.put(cs, cs);
                    }
                    else if (c instanceof String)
                    {
                        String cs = c.toString();
                        addInterfaces.put(cs, cs);
                    }
                }
            }
        }
    }

    ClassWriter cw = new ClassWriter(_componentClassName);

```

```

cw.setSuperclass(_baseClass);
for (Iterator i = addInterfaces.keySet().iterator(); i.hasNext();)
{
    String toAdd = (String)i.next();
    cw.addImplements(toAdd);
}
cw.beginClass();
try
{
    // Need to declare constructor if sub-class constructor
    // may throw exceptions.
    Constructor constructor = _baseClass.getDeclaredConstructor(new Class[0]);
    MethodWriter mw = cw.newConstructor(constructor);
    mw.beginMethod();
    NoAutoInitAttribute noAutoInit = (NoAutoInitAttribute)_attributes.findAttribute(NoAutoInitAttribute.class);
    if (noAutoInit == null)
    {
        Method init = getMethod("$init", EMPTY_CLASS_ARRAY);
        if (init != null)
        {
            mw.call("this.$init");
        }
        else
        {
            init = getMethod("init", EMPTY_CLASS_ARRAY);
            if (init != null
                && ! javax.servlet.GenericServlet.class.isAssignableFrom(_baseClass))
            {
                mw.call("this.init");
            }
        }
        mw.endMethod();
    }
}
catch (Exception ignore)
{
}
List methods = getMethods();
for (Iterator i = _attributes.iterator(); i.hasNext();)
{
    Attribute a = (Attribute)i.next();
    AttributeCompiler ac = a.getCompiler();
    if (ac != null)
    {
        ac.compile(this, cw);
    }
    if (a.getFieldName() != null)
    {
        Field field = getField(a.getFieldName());
        if (field == null)

```

```

    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnFieldNotFoundForAttribute(a.toString());
        }
    }
    else if (ac != null)
    {
        ac.compile(this, field, cw);
    }
}
if (a.getMethodName() != null)
{
    boolean match = false;
    for (Iterator j = methods.iterator(); j.hasNext(); )
    {
        Method method = (Method)j.next();
        if (a.matches(method))
        {
            match = true;
            break;
        }
    }
    if (! match && ! a.okIfMethodNotFound())
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnMethodNotFoundForAttribute(a.toString());
        }
    }
}
}
for (Iterator i = methods.iterator(); i.hasNext(); )
{
    Method method = (Method)i.next();
    int m = method.getModifiers();
    boolean ignoreMethod = false;
    if (Modifier.isStatic(m) && ! method.getName().equals("getInstance"))
    {
        if (_attributes.findAttribute(IgnoreStaticMethodsAttribute.class) == null)
        {
            if (! _bootstrap)
            {
                ComponentLog.getInstance(_baseClass).warnMethodIsStatic(method.getName());
            }
        }
        ignoreMethod = true;
    }
    if (Modifier.isFinal(m))
    {

```

```

    if (_attributes.findAttribute(IgnoreFinalMethodsAttribute.class) == null)
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnMethodIsFinal(method.getName());
        }
    }
    ignoreMethod = true;
}
if (Modifier.isPrivate(m))
{
    if (_attributes.findAttribute(IgnorePrivateMethodsAttribute.class) == null)
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnMethodIsPrivate(method.getName());
        }
    }
    ignoreMethod = true;
}
if (! Modifier.isPrivate(m) && ! Modifier.isProtected(m) && ! Modifier.isPublic(m))
{
    if (_attributes.findAttribute(IgnorePackagePrivateMethodsAttribute.class) == null)
    {
        if (! _bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnMethodIsPackagePrivate(method.getName());
        }
    }
    ignoreMethod = true;
}
if (ignoreMethod)
{
    continue;
}
boolean abstractMethod = (m & Modifier.ABSTRACT) != 0;
MethodWriter mw = cw.newMethod(method);
AttributeList methodAttributes = getAttributes(method, mw);
mw.setAttributes(methodAttributes);
if (abstractMethod)
{
    mw.setAbstract();
}
mw.setFinalParameters();
mw.beginMethod();
mw.setEmpty(true);
int attrCount = methodAttributes.size();
AttributeCompiler[] aspectCompilers = new AttributeCompiler[attrCount];
Class returnType = method.getReturnType();
if (returnType != void.class)

```

```

{
    mw.newResult();
}
mw.setEmpty(true);
Iterator iterator = methodAttributes.iterator();
for (int attrIndex = 0; attrIndex < attrCount; attrIndex++)
{
    Attribute a = (Attribute)iterator.next();
    if (a.getMethodName() != null && ! a.matches(method))
    {
        continue;
    }
    AttributeCompiler ac = a.getCompiler();
    if (ac != null && ac.ignoreMethod(this, method))
    {
        continue;
    }
    aspectCompilers[attrIndex] = ac;
    if (ac != null)
    {
        ac.beforeSuperCall(this, method, mw);
    }
}
String setResult = "";
if (returnType != void.class)
{
    setResult = mw.getResult() + " = ";
}
if (abstractMethod)
{
    for (int attrIndex = 0; attrIndex < attrCount; attrIndex++)
    {
        AttributeCompiler ac = aspectCompilers[attrIndex];
        if (ac != null)
        {
            ac.abstractBody(this, method, mw);
        }
    }
}
else if (mw.getSuperCall())
{
    String outerClassPrefix = "";
    if (mw.getInnerClassLevel() > 0)
    {
        outerClassPrefix = "_componentClassName + ".";
    }
    boolean saveEmpty = mw.isEmpty();
    mw.writeln(setResult + outerClassPrefix + "super." + mw.getName()
        + mw.getParameterList() + ";");
    mw.setEmpty(saveEmpty);
}

```

```

    }
    for (int attrIndex = attrCount - 1; attrIndex >= 0; attrIndex--)
    {
        AttributeCompiler ac = aspectCompilers[attrIndex];
        if (ac != null)
        {
            ac.afterSuperCall(this, method, mw);
        }
    }
    if (mw.isAbstract())
    {
        // Discard generated code, no attribute provided body for
        // abstract method.
        mw.setEmpty(true);
        mw.endMethod();
        if (!_bootstrap)
        {
            ComponentLog.getInstance(_baseClass).warnMethodNotImplemented(getSignature(method));
        }
        mw = cw.newMethod(method);
        mw.beginMethod();
        mw.throwRuntimeException("new com.sybase.djc.MethodNotImplementedException()");
        mw.endMethod();
        continue;
    }
    else if (mw.isEmpty() && mw.getSuperCall() && !abstractMethod)
    {
        // Discard generated code, as it only delegates to superclass.
        mw.setEmpty(true);
        mw.endMethod();
        continue;
    }
    else if (returnType != void.class)
    {
        mw.setResult();
    }
    if (abstractMethod)
    {
        // Generate empty method bodies for "instantiated" abstract methods
        mw.setEmpty(false);
    }
    mw.endMethod();
}
cw.endClass();
return cw;
}
/**
** Get all declared and public (inherited) methods in baseClass, avoiding
** any duplicates, and ensuring that for overridden methods, the
** definition from the most-derived class is included, i.e. if a base

```

**\*\* class has a method which throws an exception, and a sub-class omits**

**\*\* the throws clause, we want to omit the throws clause.**

**\*\*/**

```
public static List getMethods(Class inClass)
{
    List methods = new LinkedList();
    Method[] declaredMethods = inClass.getDeclaredMethods();
    HashMap sigHash = new HashMap();
    for (int i = 0; i < declaredMethods.length; i++)
    {
        Method method = declaredMethods[i];
        int m = method.getModifiers();
        if (Modifier.isStatic(m)
            && (method.getName().equals("getInstance")
                || method.getName().equals("$getInstance")))
        {
            // Allow factory pattern without producing warning messages.
            continue;
        }
        if (method.getDeclaringClass() == Object.class
            || (Modifier.isFinal(m) || Modifier.isStatic(m)))
        {
            // Suppress these methods as they cannot ever be overridden.
            continue;
        }
        String ms = getSignature(method);
        if (sigHash.get(ms) == null)
        {
            sigHash.put(ms, ms);
            methods.add(method);
        }
    }
    Class sc = inClass.getSuperclass();
    if (sc != null)
    {
        List scMethods = getMethods(sc);
        for (Iterator j = scMethods.iterator(); j.hasNext();)
        {
            Method scm = (Method)j.next(), method;
            String ms = getSignature(scm);
            if (sigHash.get(ms) == null)
            {
                sigHash.put(ms, ms);
                // check if this method was inherited, but overridden
                try
                {
                    {
                        method = inClass.getDeclaredMethod(scm.getName(), scm.getParameterTypes());
                    }
                }
                catch (Exception ex)
                {

```

```

        method = scm;
    }
    methods.add(method);
}
}
}
Class[] interfaces = inClass.getInterfaces();
for (int i = 0; i < interfaces.length; i++)
{
    sc = interfaces[i];
    List scMethods = getMethods(sc);
    for (Iterator j = scMethods.iterator(); j.hasNext();)
    {
        Method scm = (Method)j.next(), method;
        String ms = getSignature(scm);
        if (sigHash.get(ms) == null)
        {
            sigHash.put(ms, ms);
            // check if this method was inherited, but overridden
            try
            {
                method = inClass.getDeclaredMethod(scm.getName(), scm.getParameterTypes());
            }
            catch (Exception ex)
            {
                method = scm;
            }
            methods.add(method);
        }
    }
}
return methods;
}

public static String getSignature(Method method)
{
    StringBuffer sb = new StringBuffer(80);
    sb.append(method.getName());
    sb.append('(');
    Class[] parameters = method.getParameterTypes();
    int n = parameters.length;
    for (int i = 0; i < n; i++)
    {
        if (i > 0)
        {
            sb.append(",");
        }
        sb.append(parameters[i].getName());
    }
    sb.append(')');
    return sb.toString();
}

```



```

}
// messages for use by attribute compilers
public void errorExpectedReturnType(Method method, String returnType, Attribute attribute)
{
    if (!_bootstrap)
    {
        ComponentLog.getInstance(_baseClass).errorExpectedReturnType(getSignature(method), returnType,
attribute);
    }
}
public void warnParameterNotFound(Method method, String parameter, Attribute attribute)
{
    if (!_bootstrap)
    {
        ComponentLog.getInstance(_baseClass).warnParameterNotFound(getSignature(method), parameter,
attribute);
    }
}
public void warnParameterNotFound(MethodWriter method, String parameter, Attribute attribute)
{
    if (!_bootstrap)
    {
        ComponentLog.getInstance(_baseClass).warnParameterNotFound(method.getSignature(), parameter,
attribute);
    }
}
public void warnParameterNotUsed(Method method, String parameter, Attribute attribute)
{
    if (!_bootstrap)
    {
        ComponentLog.getInstance(_baseClass).warnParameterNotUsed(getSignature(method), parameter,
attribute);
    }
}
public void warnParameterNotUsed(MethodWriter method, String parameter, Attribute attribute)
{
    if (!_bootstrap)
    {
        ComponentLog.getInstance(_baseClass).warnParameterNotUsed(method.getSignature(), parameter,
attribute);
    }
}
// private methods
/**
 ** Check for bootstrap components that can't be dynamically recompiled.
 **/
private boolean allowRuntimeCompile(Class bc)
{
    if (bc == ComponentLog.class
        || bc == ProcessUtil.class)

```

```

    {
        return false;
    }
    if (bc.getName().startsWith("com.sybase.djc.log."))
    {
        return false;
    }
    return true;
}
private void compile()
{
if (allowRuntimeCompile(_baseClass))
{
    if (_debug)
    {
        System.out.println("DEBUG: djc " + getName());
    }
    getClassWriter().compile(_baseClass);
}
}
private Object evaluateComponentClass()
{
boolean compiled = false;
if (_recompile)
{
    compile();
    compiled = true;
}
try
{
    return loadClass(_componentClassName);
}
catch (Exception ex)
{
    if (!compiled)
    {
        compile();
    }
    return loadClass(_componentClassName);
}
}
private Object evaluateSingleton()
{
try
{
    return newInstance();
}
catch (RuntimeException ex)
{
    throw (RuntimeException)ex;
}
}

```

```

    }
    catch (Exception ex)
    {
        throw new SystemException(ex);
    }
}

private Class getComponentClass()
{
    return (Class)_componentClass.getValue();
}

private Object getSingleton()
{
    return _singleton.getValue();
}

private Class loadClass(String className)
{
    return ThreadContext.loadClass(className, _baseClass);
}
}

// ComponentAttributes.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
package com.sybase.djc;
import com.sybase.djc.attributes.*;
import com.sybase.djc.security.*;
import com.sybase.djc.sql.JdbcType;
import com.sybase.djc.sql.VerifyType;
import com.sybase.djc.transaction.IsolationLevel;
import com.sybase.djc.transaction.TransactionType;
/**
 ** Attribute factory for all pre-defined attributes.
 **/

public abstract class ComponentAttributes
{
    // constants
    public static final IsolationLevel ReadUncommitted = IsolationLevel.ReadUncommitted;
    public static final IsolationLevel ReadCache = IsolationLevel.ReadCache;
    public static final IsolationLevel ReadCacheVerifyUpdates = IsolationLevel.ReadCacheVerifyUpdates;
    public static final IsolationLevel ReadCommitted = IsolationLevel.ReadCommitted;
    public static final IsolationLevel ReadCommittedWithCache = IsolationLevel.ReadCommittedWithCache;
    public static final IsolationLevel ReadCommittedVerifyUpdates = IsolationLevel.ReadCommittedVerifyUpdates;
    public static final IsolationLevel ReadCommittedVerifyUpdatesWithCache =
IsolationLevel.ReadCommittedVerifyUpdatesWithCache;
    public static final IsolationLevel RepeatableRead = IsolationLevel.RepeatableRead;
    public static final IsolationLevel RepeatableReadWithCache = IsolationLevel.RepeatableReadWithCache;
    public static final IsolationLevel Serializable = IsolationLevel.Serializable;
    public static final IsolationLevel SerializableWithCache = IsolationLevel.SerializableWithCache;
    public static final JdbcType ARRAY = JdbcType.ARRAY;
    public static final JdbcType BIGINT = JdbcType.BIGINT;
    public static final JdbcType BINARY = JdbcType.BINARY;
    public static final JdbcType BIT = JdbcType.BIT;

```

```

public static final JdbcType BLOB = JdbcType.BLOB;
// JDK 1.4 only: public static final JdbcType BOOLEAN = JdbcType.BOOLEAN;
public static final JdbcType CHAR = JdbcType.CHAR;
public static final JdbcType CLOB = JdbcType.CLOB;
// JDK 1.4 only: public static final JdbcType DATALINK = JdbcType.DATALINK;
public static final JdbcType DATE = JdbcType.DATE;
public static final JdbcType DECIMAL = JdbcType.DECIMAL;
public static final JdbcType DISTINCT = JdbcType.DISTINCT;
public static final JdbcType DOUBLE = JdbcType.DOUBLE;
public static final JdbcType FLOAT = JdbcType.FLOAT;
public static final JdbcType INTEGER = JdbcType.INTEGER;
public static final JdbcType JAVA_OBJECT = JdbcType.JAVA_OBJECT;
public static final JdbcType LONGVARBINARY = JdbcType.LONGVARBINARY;
public static final JdbcType LONGVARCHAR = JdbcType.LONGVARCHAR;
public static final JdbcType NULL = JdbcType.NULL;
public static final JdbcType NUMERIC = JdbcType.NUMERIC;
public static final JdbcType OTHER = JdbcType.OTHER;
public static final JdbcType REAL = JdbcType.REAL;
public static final JdbcType REF = JdbcType.REF;
public static final JdbcType SMALLINT = JdbcType.SMALLINT;
public static final JdbcType STRUCT = JdbcType.STRUCT;
public static final JdbcType TIME = JdbcType.TIME;
public static final JdbcType TIMESTAMP = JdbcType.TIMESTAMP;
public static final JdbcType TINYINT = JdbcType.TINYINT;
public static final JdbcType VARBINARY = JdbcType.VARBINARY;
public static final JdbcType VARCHAR = JdbcType.VARCHAR;
public static final TransactionType Ignored = TransactionType.Ignored;
public static final TransactionType Required = TransactionType.Required;
public static final TransactionType RequiresNew = TransactionType.RequiresNew;
public static final TransactionType Supported = TransactionType.Supported;
public static final TransactionType NotSupported = TransactionType.NotSupported;
public static final TransactionType Mandatory = TransactionType.Mandatory;
public static final TransactionType Never = TransactionType.Never;
// attribute factory methods
public static NoAutoInitAttribute NoAutoInit()
{
    return new NoAutoInitAttribute();
}
public static IgnoreFinalMethodsAttribute IgnoreFinalMethods()
{
    return new IgnoreFinalMethodsAttribute();
}
public static IgnorePackagePrivateMethodsAttribute IgnorePackagePrivateMethods()
{
    return new IgnorePackagePrivateMethodsAttribute();
}
public static IgnorePrivateMethodsAttribute IgnorePrivateMethods()
{
    return new IgnorePrivateMethodsAttribute();
}

```

```
public static final IgnoreStaticMethodsAttribute IgnoreStaticMethods()
{
    return new IgnoreStaticMethodsAttribute();
}
public static DisplayNameAttribute DisplayName()
{
    return new DisplayNameAttribute();
}
public static DisplayNameAttribute DisplayName(String name)
{
    return new DisplayNameAttribute().value(name);
}
public static DescriptionAttribute Description()
{
    return new DescriptionAttribute();
}
public static DescriptionAttribute Description(String text)
{
    return new DescriptionAttribute().value(text);
}
public static SharedObjectAttribute SharedObject()
{
    return new SharedObjectAttribute();
}
public static DelegateAbstractMethodsAttribute DelegateAbstractMethods()
{
    return new DelegateAbstractMethodsAttribute();
}
public static FormatMethodAttribute FormatMethod(String methodSignature)
{
    return new FormatMethodAttribute(methodSignature);
}
public static LogMethodAttribute LogMethod(String methodSignature)
{
    return new LogMethodAttribute(methodSignature);
}
public static LogApplicationExceptionsAttribute LogApplicationExceptions()
{
    return new LogApplicationExceptionsAttribute();
}
public static LogApplicationExceptionsAttribute LogApplicationExceptions(String methodSignature)
{
    return new LogApplicationExceptionsAttribute(methodSignature);
}
public static LogSystemExceptionsAttribute LogSystemExceptions()
{
    return new LogSystemExceptionsAttribute();
}
public static LogSystemExceptionsAttribute LogSystemExceptions(String methodSignature)
{

```

```

    return new LogSystemExceptionsAttribute(methodSignature);
}
public static TraceAttribute TraceMethod(String methodSignature)
{
    return new TraceAttribute(methodSignature);
}
public static TraceAttribute TraceObjectMethods()
{
    return new TraceAttribute().methods(Component.ObjectMethods);
}
public static TraceAttribute TraceProtectedMethods()
{
    return new TraceAttribute().methods(Component.ProtectedMethods);
}
public static TraceAttribute TracePublicMethods()
{
    return new TraceAttribute().methods(Component.PublicMethods);
}
public static MessageListenerAttribute MessageListener()
{
    return new MessageListenerAttribute();
}
public static NamingContextAttribute NamingContext()
{
    return new NamingContextAttribute();
}
public static NamingContextAttribute NamingContext(Class contextClass)
{
    return new NamingContextAttribute().value(contextClass);
}
public static RemoteInterfaceAttribute RemoteInterface()
{
    return new RemoteInterfaceAttribute();
}
public static RemoteInterfaceAttribute RemoteInterface(Class remoteInterface)
{
    return new RemoteInterfaceAttribute().value(remoteInterface);
}
public static ProfileAttribute ProfileMethod(String methodSignature)
{
    return new ProfileAttribute(methodSignature);
}
public static ProfileAttribute ProfileObjectMethods()
{
    return new ProfileAttribute().methods(Component.ObjectMethods);
}
public static ProfileAttribute ProfileProtectedMethods()
{
    return new ProfileAttribute().methods(Component.ProtectedMethods);
}
}

```

```
public static ProfileAttribute ProfilePublicMethods()
{
    return new ProfileAttribute().methods(Component.PublicMethods);
}
public static PersistentFieldAttribute PersistentField(String fieldName)
{
    return new PersistentFieldAttribute(fieldName);
}
public static PersistentObjectAttribute PersistentObject()
{
    return new PersistentObjectAttribute();
}
public static PrimaryKeyClassAttribute PrimaryKeyClass()
{
    return new PrimaryKeyClassAttribute();
}
public static PrimaryKeyClassAttribute PrimaryKeyClass(Class keyClass)
{
    return new PrimaryKeyClassAttribute().value(keyClass);
}
public static PrimaryKeyFieldAttribute PrimaryKeyField(String fieldName)
{
    return new PrimaryKeyFieldAttribute(fieldName);
}
public static ForeignKeyFieldAttribute ForeignKeyField(String fieldName)
{
    return new ForeignKeyFieldAttribute(fieldName);
}
public static QueryMethodAttribute QueryMethod(String methodSignature)
{
    return new QueryMethodAttribute(methodSignature);
}
public static PermitAccessAttribute PermitAccess()
{
    return new PermitAccessAttribute();
}
public static PermitAccessAttribute PermitAccess(String methodSignature)
{
    return new PermitAccessAttribute(methodSignature);
}
public static DenyAccessAttribute DenyAccess()
{
    return new DenyAccessAttribute();
}
public static DenyAccessAttribute DenyAccess(String methodSignature)
{
    return new DenyAccessAttribute(methodSignature);
}
public static RunAsAttribute RunAs()
{

```

```

        return new RunAsAttribute();
    }
    public static RunAsAttribute RunAs(String methodSignature)
    {
        return new RunAsAttribute(methodSignature);
    }
    public static SecureObjectAttribute SecureObject()
    {
        return new SecureObjectAttribute();
    }
    public static SecurityActionAttribute SecurityAction(String action)
    {
        return new SecurityActionAttribute(action);
    }
    public static SecurityResourceAttribute SecurityResource(String resource)
    {
        return new SecurityResourceAttribute(resource);
    }
    public static ThreadMonitorAttribute ThreadMonitor()
    {
        return new ThreadMonitorAttribute();
    }
    public static ThreadMonitorAttribute ThreadMonitor(String methodSignature)
    {
        return new ThreadMonitorAttribute(methodSignature);
    }
    public static TransactionAttribute Transaction()
    {
        return new TransactionAttribute();
    }
    public static TransactionAttribute Transaction(String methodSignature)
    {
        return new TransactionAttribute(methodSignature);
    }
    public static WebApplicationAttribute WebApplication()
    {
        return new WebApplicationAttribute();
    }
    public static WebComponentAttribute WebComponent()
    {
        return new WebComponentAttribute();
    }
}

// MockObject.java
// Copyright (c) 2004. Sybase, Inc. All Rights Reserved.
package com.sybase.djc;
import com.sybase.djc.compiler.*;
import java.lang.reflect.*;
import java.util.*;
public interface MockObject

```



```

{
    public static final Attribute[] attributes =
    {
        new MockAttribute()
    };
    public static class MockAttribute extends Attribute
    {
        public AttributeCompiler getCompiler()
        {
            return new MockCompiler();
        }
    }
    public static class MockCompiler extends AttributeCompiler
    {
        public void compile(Component component, ClassWriter cw)
        {
            cw.newInstanceField(MockDelegate.class, "_delegate", "new com.sybase.djc.MockObject.MockDelegate()");
        }
        public void beforeSuperCall(Component component, Method method, MethodWriter mw)
        {
            if (mw.isAbstract())
            {
                mw.notAbstract();
            }
            else
            {
                // Mock Objects do not delegate to superclass.
                mw.setSuperCall(false);
            }
            if (Component.isObjectMethod(method))
            {
                if (method.getName().equals("toString"))
                {
                    // Don't allow expected calls for toString(), as this tends
                    // to produce undesirable side effects when unit testing or
                    // tracing frameworks call toString(). If the need arises
                    // to add an expected call for toString(), it is recommended
                    // to add an additional method to which toString() delegates
                    // (e.g. getName()), and add expected calls for that method
                    // instead.
                    mw.setResult(mw.string(component.getBaseClass().getName() + ":",
                        + " + _delegate.toString()");
                    return;
                }
                if (method.getName().equals("finalize"))
                {
                    // Best avoided also, as GC is unpredictable and allowing
                    // finalize as an expected call would be fairly unreliable.
                    return;
                }
            }
        }
    }
}

```

```

    }
    boolean plainMethod = true;
    try
    {
        MockObject.class.getMethod(method.getName(), method.getParameterTypes());
        plainMethod = false;
    }
    catch (NoSuchMethodException ignore)
    {
    }
    ParameterList pl = mw.getParameterList();
    if (!plainMethod)
    {
        if (method.getReturnType() == void.class)
        {
            mw.call("_delegate." + method.getName(), pl);
        }
        else
        {
            mw.setResult(mw.invoke("_delegate." + method.getName(), pl));
        }
    }
    else
    {
        mw.beginTry();
        String expr = mw.invoke("_delegate.invoke",
                                mw.string(method.getName()),
                                pl.getClassArray(),
                                pl.getObjectArray());
        if (method.getReturnType() == void.class)
        {
            mw.statement(expr);
        }
        else if (method.getReturnType().isPrimitive())
        {
            mw.setResult(mw.unwrap(mw.getReturnType(), expr));
        }
        else
        {
            mw.setResult(mw.cast(mw.getReturnType(), expr));
        }
        LocalVariable ex = mw.catchException();
        mw.throwException(ex);
        mw.endTry();
    }
}

}

public static class MockDelegate
{
    private int _callCount = 0;

```

```

private int _groupIndex = 0;
private LinkedList _expectedCalls = new LinkedList();
public Object invoke(String methodName, Class[] types, Object[] parameters) throws Exception
{
    for (;;)
    {
        int countAtCurrentIndex = 0;
        int nextGroupIndex = -1;
        for (Iterator i = _expectedCalls.iterator(); i.hasNext(); )
        {
            ExpectedCall ec = (ExpectedCall)i.next();
            if (ec.getCallCount() == ec.getMaximumCount())
            {
                // This expected call is no longer active.
                continue;
            }
            int index = ec.getGroupIndex();
            if (index == _groupIndex)
            {
                countAtCurrentIndex++;
            }
            else if (index < _groupIndex)
            {
                // This expected call's group is no longer active.
                continue;
            }
            else if (index > _groupIndex)
            {
                // This expected call is waiting for a later group of calls.
                // Retain the minimum next group index for next iteration
                // of outer loop.
                if (nextGroupIndex == -1 || index < nextGroupIndex)
                {
                    nextGroupIndex = index;
                }
                continue;
            }
        }
        try
        {
            Object callObject = ec.getCallObject();
            Method method = callObject.getClass().getMethod(methodName, types);
            method.setAccessible(true);
            if (method.getDeclaringClass() == Object.class)
            {
                // Calls to java.lang.Object methods must be
                // handled with addExpectedCall. Otherwise
                // the above can result in ambiguity, since
                // any java.lang.Object method can be found
                // in any ExpectedCall object.
                throw new NoSuchMethodException();
            }
        }
    }
}

```

```

    }
    ec.addCall();
    _callCount++;
    try
    {
        Object result = method.invoke(callObject, parameters);
        return result;
    }
    catch (InvocationTargetException ex)
    {
        Throwable target = ex.getTargetException();
        if (target instanceof Exception)
        {
            throw (Exception)target;
        }
        else if (target instanceof Error)
        {
            throw (Error)target;
        }
        else
        {
            throw new SystemException(target.toString());
        }
    }
}
catch (NoSuchMethodException ignore)
{
}
}
if (countAtCurrentIndex > 0)
{
    // Other calls were expected in this group.
    // Break to allow exception to be thrown.
    break;
}
if (nextGroupIndex == -1)
{
    // No more call groups were expected.
    // Break to allow exception to be thrown.
    break;
}
_groupIndex = nextGroupIndex;
}
String calls = getExpectedCalls();
if (calls.length() > 0)
{
    throw new VerifyExpectedCallsError("unexpected call: " + methodName + getSignature(types)
        + ": expected " + calls);
}
else

```

```

    {
        throw new VerifyExpectedCallsError("unexpected call: " + methodName + getSignature(types)
            + ": expected " + (_callCount > 0 ? "no more calls" : "no calls"));
    }
}

public ExpectedCall addExpectedCall(Object callObject)
{
    ExpectedCall ec = new ExpectedCall(callObject);
    _expectedCalls.add(ec);
    return ec;
}

public void verifyExpectedCalls()
{
    String calls = getExpectedCalls();
    if (calls.length() > 0)
    {
        String s = calls.indexOf(';') == -1 ? "" : "s";
        throw new VerifyExpectedCallsError("missing expected call" + s + ": " + calls);
    }
}

private String getExpectedCalls()
{
    StringBuffer calls = new StringBuffer();
    for (Iterator i = _expectedCalls.iterator(); i.hasNext(); )
    {
        ExpectedCall ec = (ExpectedCall)i.next();
        if (ec.getCallCount() >= ec.getMinimumCount())
        {
            // This expected call has been minimally satisfied.
            continue;
        }
        if (ec.getCallCount() == ec.getMaximumCount())
        {
            // This expected call has been maximally satisfied.
            continue;
        }
        if (calls.length() > 0)
        {
            calls.append("; ");
        }
        calls.append(getExpectedMethods(ec));
    }
    return calls.toString();
}

private String getExpectedMethods(ExpectedCall ec)
{
    StringBuffer expected = new StringBuffer();
    Method[] methods = ec.getCallObject().getClass().getMethods();
    for (int i = 0; i < methods.length; i++)
    {

```

```

        Method method = methods[i];
        if (method.getDeclaringClass() == Object.class)
        {
            continue;
        }
        if (expected.length() > 0)
        {
            expected.append(" or ");
        }
        expected.append(method.getName());
        expected.append(getSignature(method.getParameterTypes()));
    }
    if (expected.length() == 0)
    {
        return "[no methods in call object - please check usage of addExpectedCall]";
    }
    return expected.toString();
}

private String getSignature(Class[] types)
{
    StringBuffer sig = new StringBuffer("");
    for (int i = 0; i < types.length; i++)
    {
        Class t = types[i];
        if (i > 0)
        {
            sig.append(',');
        }
        sig.append(t.getName());
    }
    sig.append(")");
    return sig.toString();
}

}

// public methods available in all mock objects
public ExpectedCall addExpectedCall(Object expectedCall);
public void verifyExpectedCalls();
}

```